Final Project Routing Report

Hongchen Li
Electrical Engineering Department
University of California, Los Angeles
Los Angeles, U.S.A
hongchenli@ucla.edu

***Abstract*—Given a set of terminals and a set of obstacles on a plane, two designs of algorithms are proposed to connect these terminals and to avoid running through any obstacle to construct a tree with a possible minimal total wirelength. Design I uses the breadth-first search (BFS) to find the shortest path between any two given terminals and then applies Prim's algorithm for the minimum spanning tree (MST). Design II uses the BFS and then applies Kruskal's algorithm for the MST. Design I and design II achieved the total wirelength of 20 for the example "init", 176 for the test file "tb1", and 238 and 248 for the file "tb2" separately. Theoretical optimality are presented.**

Keywords—Routing; Prim; Krustal; obstacle; breadth-first search (BFS); minimum spanning tree (MST)

I. INTRODUCTION

Minimum Spanning Tree (MST) problem is to find a min weight set of edges that connects all of the vertices, given connected graph with positive edge weights. MST is fundamental problem with diverse applications, such as network design and approximation algorithms for NP-hard problems[1]. To solve the NP-hard problem in this project, whose goal is to connect all the terminals without running through any obstacle, the fundamental solution can apparently be one case of the MST.

One efficient method to solve the MST problem is Greedy algorithm. Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit[2]. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: At every step, one can make a choice that looks best at the moment, and get the optimal solution of the complete problem.

If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. Following are some standard algorithms that are Greedy algorithms.

- Kruskal's MST: In Kruskal's algorithm, one creates a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.
- Prim's Minimum Spanning Tree: In Prim's algorithm also, one creates a MST by picking edges one by one. Two sets need to be maintained: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

Greedy are applied in this project provided that problem routing can be seen as advanced MST. Two above greedy algorithm designs which provide solutions for this problem are presented separately. Design I uses the breadth-first search (BFS) to find the shortest path between any two given terminals and then applies Prim's algorithm for the MST. Design II also uses the BFS and then applies Kruskal's algorithm for the MST. In Section 2, the design of Prim's algorithm is presented. In Section 3, the other design of Kruskal's algorithm is described. Experimental results are analyzed and theoretical optimality are discussed in Section 4 and the discussion is finally concluded in Section 5.

II. DESIGN OF PRIM'S ALGROTHIM

A. *Prim's Algorithm*

The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree. Follow the steps below and grow a tree as shown in Fig. 1. The key idea is to expand the current tree by adding the lightest (shortest) edge leaving it and its endpoint.

- Step 0: Choose any element r; set S = {r} and A = ∅.
- Step 1: Find a lightest edge such that one endpoint is in S and the other is in V \ S. Add this edge to A and its (other) endpoint to S.
- Step 2: If V \ S = ∅, then stop and output (minimum) spanning tree (S, A). Otherwise go to Step 1.

```
Prim's algorithm:
let S be a single vertex r, A empty
while (S has fewer vertices than V) {
        find the lightest edge connecting S to V-S
        add it to S, add this edge to A}
```

Fig. 1. Prim's algorithm

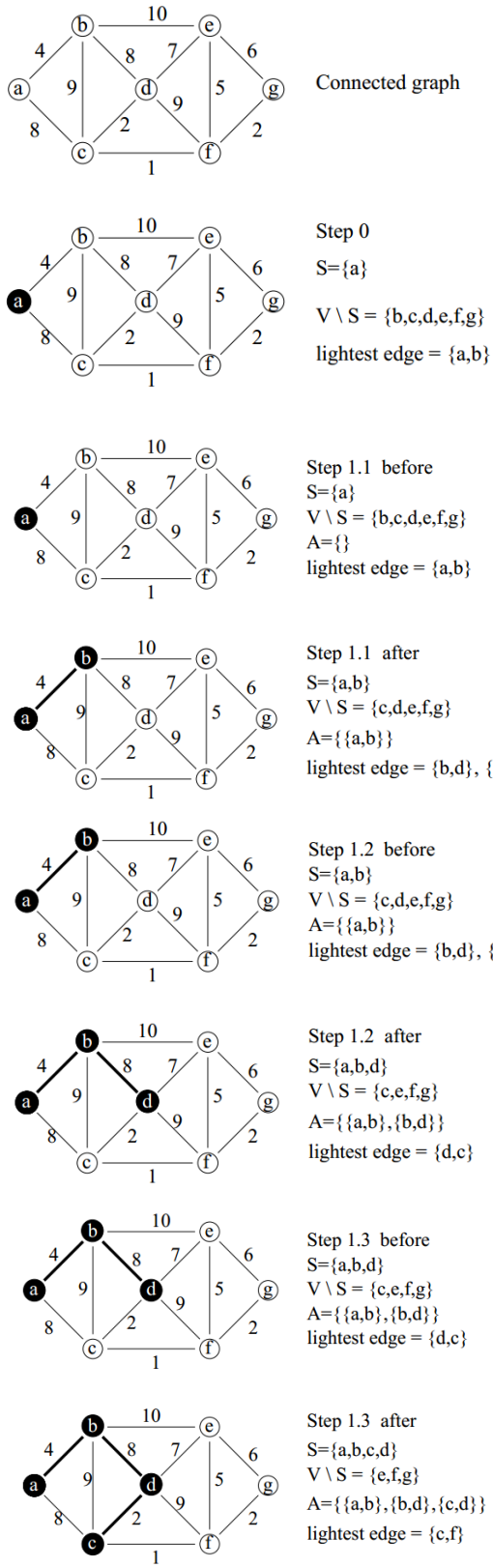


Fig. 2. Prim's Algorithm to grow a MST[3]

B. *C++ Implementation on Prim's Algorithm*

In the declaration, class Point is set with three components: x, y, paths, first two of which represent position and the last one of which is a vector of path, which is a vector of Point. In the Prim's algorithm design, each terminal is set with its own position and the paths from itself to any other terminal. To simplify the process, the paths are also sorted in a set. "The Smaller path" is defined as a path with fewer points. If two path lengths are equal, compare their first or last point position so that the paths with the same lengths but between different terminals will not be discarded. The design starts with terminals from set V, then compare each terminal's first path, i.e. the smallest path in each sorted order, and pick the smallest one. If the endpoint of this path is not in V, put this endpoint into V and put this path into the path vector "mst". Erase this path from its terminal. Otherwise, pick the next smallest path and repeat the above process. When the size of set V equals to that of set U, which contains all terminals, Prim's algorithm ends. Therefore, the MST connecting each terminal is attained. More details can be found in Fig. 3.

```
//Prim
  PathSet mst_temp;
  int index;
     while( U->size() != V->size() ){
   // start with terms in V
        for(PointSet::iterator psp1 = V->begin();
psp1 !=V->end(); psp1++){
             Point* point_t = (*psp1);
             index = termIndex(point_t);
//search mininal edges whose terminals do not belong to V
             for(PathSet::iterator spe = Edges[index].begin();
spe !=Edges[index].end(); spe++){
                    Path* min = (*spe);
                    Point* term = new Point((*min)[0]);
// if terminal doesn't belong to V, put this path into temp mst
                    if(!findTerms_V(term, V)){
                       mst_temp.insert(min);
                       break;
                    }
                    else
                       Edges[index].erase(*spe);
             }
      }

          if( mst_temp.size() != 0){
             // put the first(min) path into mst
                PathSet::iterator msp = mst_temp.begin();
                Path* msf= (*msp);
                mst.push_back(msf);
             // put terminal into V
                Point* termV = new Point((*msf)[0]);
                V->insert(termV);
             // clear mst_temp
                while(mst_temp.size() != 0){
                   PathSet::iterator msc = mst_temp.begin();
                   mst_temp.erase(*msc);}
                 }
```

Fig. 3. C++ Implementation on Prim's Algorithm

```
procedure BFS(G,s) is
    let q be a queue
    q.push(s)
    label s as discovered
    while q is not empty
        s ← q.pop()
        for all edges from s to w in G.adjacentEdges(s) do
            if w is not labeled as discovered
                q.push(w)
                label w as discovered
```

Fig. 4 BFS Algorithm

```
Kruskal's algorithm:
sort the edges of  V in increasing order by length
keep a subgraph S of V, initially empty
for each edge e in sorted order
    if the endpoints of e are disconnected in S
    add e to S
return S
```

Fig. 6 Kruskal's algorithm

C. BFS Algorithm

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root and explores the neighbor nodes first, before moving to the next level neighbors.[4]

BFS algorithm is applied both in Design I and II in order to find the shortest path between any two terminals and is described in details in Fig.3 and Fig.4. Based on the fundamentals, BFS in this project includes two parts: trace and trace back. The trace part starts with a terminal and ends with a matrix labeled path lengths from the terminal to each empty position. The region representing the obstacles is labeled as -1, and thus will not be running through during the trace part. After the above procedures, trace from other terminals back to the first terminal and record the points run through into a path in order to save in the first terminal. By repeating the above process, each single terminal carries its shortest paths from itself to any other terminals.

III. DESIGN OF KRUSKAL ALGROTHIM

Since this design also uses BFS algorithm, which can be referred to in Section 2, to find the shortest path between any two terminals, it's better to directly skip to the Kruskal's algorithm.

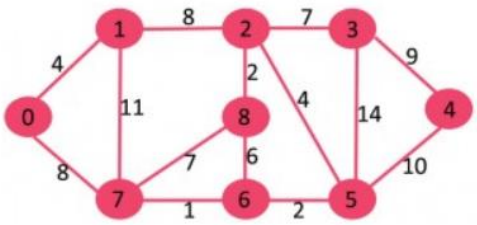
A. Kruskal's Algorithm

Kruskal's algorithm can be briefly summarized as the following. Assuming V is the number of vertices in the given graph, firstly sort all the edges in non-decreasing order of their weight, and then pick the smallest edge and check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it. Repeat the picking edge step until there are (V-1) edges in the spanning tree. Above steps are briefly described in Fig.6[5] and are then presented in details, as shown in Fig.7[6].

```
    void BFS(Point* s){
      //TRACE
      int dir[4][2] = {0,1,0,-1,1,0,-1,0};
      queue<Point*> q;
      Point* cur = new Point();
      int cx,cy,nx,ny;
     //create trace[][] to maze
      int trace[upperright->x+2][upperright->y+2];

      for (int j = upperright->y; j > 0; j--) {
        for (int i = 1; i <= upperright->x; i++) {
          trace[i][j] = 0;
        }
      }
      trace[s->x][s->y]=1;//BFS from point s
      q.push(s);
      while(!q.empty())
      {  cur = q.front();
        cx = cur->x;
        cy = cur->y;
        q.pop();
        for(int i=0; i<4; i++)  {
          nx = cx + dir[i][0];
          ny = cy + dir[i][1];
          if(trace[nx][ny] == 0 && G[nx][ny] > 0){
            trace[nx][ny] = trace[cx][cy] + 1;
            Point* nex = new Point(nx,ny);
            q.push(nex);
          }
        }
      }
      //TRACE BACK
      Point* tp = new Point();
      int tx, ty, ntx, nty, nextemp;
      for(int i = 0; i < terms.size(); i++){
          Path *sPath = new Path();
          * tp = Point(terms[i]);
          sPath->push_back(terms[i]);
          while(!point_isEqual(tp,s)){
            for(int j = 0; j < 4; j++){
              tx = tp->x ;
              ty = tp->y ;
              ntx = tx + dir[j][0];
              nty = ty + dir[j][1];
              nextemp = trace[ntx][nty];
              if (nextemp == trace[tx][ty]-1){
                Point* tbp = new Point(ntx, nty);
                sPath->push_back(tbp);
                tp->x = ntx;//move tp
                tp->y = nty;
                break; }
            }
          }
          s->paths->push_back(sPath);
      }
}
```

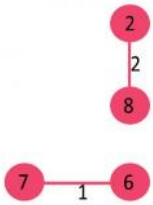
Fig. 5 BFS Implementation on C++



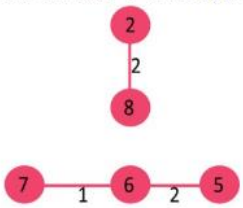
Now pick all edges one by one from sorted list of edges

1. *Pick edge 7-6:* No cycle is formed, include it.

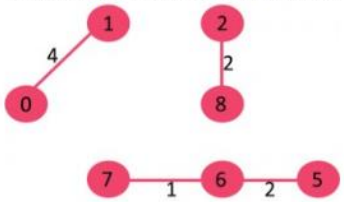
2. *Pick edge 8-2:* No cycle is formed, include it.



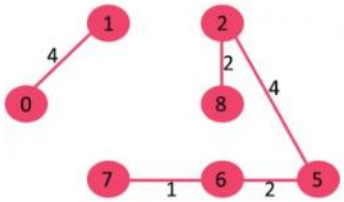
3. *Pick edge 6-5:* No cycle is formed, include it.



4. *Pick edge 0-1:* No cycle is formed, include it.

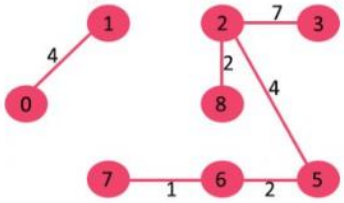


5. *Pick edge 2-5:* No cycle is formed, include it.



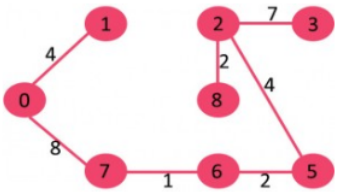
6. *Pick edge 8-6:* Since including this edge results in cycle, discard it.

7. *Pick edge 2-3:* No cycle is formed, include it.



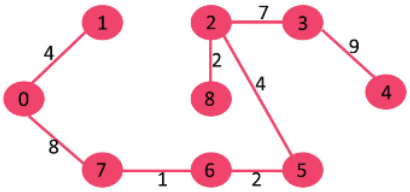
8. *Pick edge 7-8:* Since including this edge results in cycle, discard it.

9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in cycle, discard it.

11. *Pick edge 3-4:* No cycle is formed, include it.



Since the number of edges included equals (V – 1), the algorithm stops here.

Fig. 7 Kruskal's algorithm to grow a MST

B. C++ Implementation on Kruskal

Detailed codes are as shown in Fig.8. The key is to judge if adding an edge will form a cycle with the spanning tree.

The method used to judge a circle is to set a father node for every terminal. To start with, every terminal's father node is itself. If the two endpoints, e.g. a and b, of an edge have different father nodes, one can add this edge provided that this edge will not form a cycle. Then change a's father node to b's father node. By repeating the above steps in adding edges and terminals, one can finally ends with a MST connecting each terminal.

```
//Krustal C++
    father = new int[terms.size()];// father nodes
    for (int i = 0; i != terms.size(); i++){
        father[i] = i;
    }
    Path* mini = new Path();
        while ( Edges.size() != 0 ){
            PathSet::iterator spe = Edges.begin();
            mini = (*spe);
            Point* termb = new Point((*mini)[0]);
            Point*          terme          =          new
Point((*mini)[(mini->size())-1]);
            int fb = find(termIndex(termb));
            int fe = find(termIndex(terme));
          // different father node would avoid loop
            if(fb != fe){
                father[fb] = fe;
                completeGraph.push_back(mini);
                V->insert(termb);
                V->insert(terme);
            }
            Edges.erase(*spe);
    }
```

Fig. 8 Kruskal's Implementation on C++

IV. The Analysis and Discussion of Output Results

A. Output Results

The outputs for design I and design II are dumped to files named "mst_result.txt" and "routing_result.txt", seperatively. Given a simple example "init", the output results are shown in Fig. 9 below. The grid is 10*10, with four terminals and two obstacles. The total wirelength for both designs is 20. For another two test examples, "tb1" and "tb2" whose grids are both 100*100, the output graphs are too big to see on a screen and thus are better referred to from the dumped files. The wirelength for "tb1" in design I is 176, which is the same with that in design II. The wirelength for "tb2" in design I is 238, while that of design II is 248.

B. Analysis and Optimazation

1) MST and the Steiner Tree

The problem in this project is actually a NP-complete problem. In fact, Prim's and Kruskal's, which are Greedy algorithms, mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. To optimize the solution and to minimize the total wirelength, one should consider the Steiner tree instead of the MST.

The Steiner tree problem is superficially similar to the minimum spanning tree problem: given a set V of points

```
linux@ubuntu:~/Desktop/Proj1$ ./maze init
Begin ...
grid (10,10)
term (1,4)
term (6,2)
term (9,9)
term (9,2)
obs (2,2) (3,4)
obs (4,3) (5,4)
Printing out MST ...
 N |  N |  N |  N |  N |  N |  N |  N |  N |  N |
 N |  N |  N |  N |  N |  N |  N |  N |  1 |  N |
 N |  N |  N |  N |  N |  N |  N |  N |  1 |  N |
 N |  N |  N |  N |  N |  N |  N |  N |  1 |  N |
 N |  N |  N |  N |  N |  N |  N |  N |  1 |  N |
 1 |  1 |  1 |  1 |  1 |  1 |  N |  N |  1 |  N |
 1 | -1 | -1 | -1 | -1 |  1 |  N |  N |  1 |  N |
 N | -1 | -1 | -1 | -1 |  1 |  N |  N |  1 |  N |
 N | -1 | -1 |  N |  N |  1 |  1 |  1 |  1 |  N |
 N |  N |  N |  N |  N |  N |  N |  N |  N |  N |
Printing out complete graph ...
 N |  N |  N |  N |  N |  N |  N |  N |  N |  N |
 N |  N |  N |  N |  N |  N |  N |  N |  1 |  N |
 N |  N |  N |  N |  N |  N |  N |  N |  1 |  N |
 N |  N |  N |  N |  N |  N |  N |  N |  1 |  N |
 N |  N |  N |  N |  N |  N |  N |  N |  1 |  N |
 1 |  1 |  1 |  1 |  1 |  1 |  N |  N |  1 |  N |
 1 | -1 | -1 | -1 | -1 |  1 |  N |  N |  1 |  N |
 N | -1 | -1 | -1 | -1 |  1 |  N |  N |  1 |  N |
 N | -1 | -1 |  N |  N |  1 |  1 |  1 |  1 |  N |
 N |  N |  N |  N |  N |  N |  N |  N |  N |  N |
Total runtime (part1):0 usec.
Total runtime (part2):0 usec.
Prim wirelength (part1):20
Kruskal wirelength (part2):20
Finished.
linux@ubuntu:~/Desktop/Proj1$
```

Fig. 9 Results for example "init"

(vertices), interconnect them by a network (graph) of shortest length, where the length is the sum of the lengths of all edges. The difference between the Steiner tree problem and the minimum spanning tree problem is that, in the Steiner tree problem, extra intermediate vertices and edges may be added to the graph in order to reduce the length of the spanning tree. These new vertices introduced to decrease the total length of connection are known as Steiner points or Steiner vertices. It has been proved that the resulting connection is a tree, known as the Steiner tree. There may be several Steiner trees for a given set of initial vertices.

Typical algorithms for the Steiner tree can be divided into two catalogues: exact algorithms, such as GeoSteiner, and Heuristics, such as 1-Steiner. Moreover, recent progress are made on the Steiner tree problem, such as the CktSteiner, which is an obstacle avoiding Steiner minimal tree (OASMT) algorithm.

1-Steiner has been tried at first in design II and then was given up. Prim's, using an adjacency matrix graph representation and linearly searching an array of weights to find the minimum weight edge, requires $O(n^2)$ running time. Kruskal's algorithm can be shown to run in O(m log n) time, where m is the number of edges in the graph and n is the number of vertices. According to 1-Steiner, adding n Steiner points randomly and then repeating Prim's or Kruskal's n times extend the running time in exponential form, for example in the experiment, 10^3 times longer after each Kruskal's to form MST, whereas the wirelength can only be reduced by less than 10. After a tradeoff between the wirelength and running time, 1-Steiner algorithms is given up.

2) BFS and Depth-first Search (DFS)

In a maze problem, the defining characteristic of DFS is that, whenever DFS visits a maze cell c, it next searches the sub-maze whose origin is c before searching any other part of the maze. This is accomplished by using a Stack to store the nodes. The end result is that DFS will follow some path through the maze as far as it will go, until a dead end or previously visited location is found. When this occurs, the search backtracks to try another path, until it finds an exit. DFS is more memory-efficient.

For BFS, the end result is that this algorithm will visit all the cells in order of their distance from the entrance. First, it visits all locations one step away, then it visits all locations that are two steps away, and so on, until an exit is found. Although, BFS might use more memory, it has the nice property that it will naturally discover the shortest route through the maze, which exactly fits the needs of this project. Besides, BFS and DFS should be similar in terms of running time.

3) Prim's Algorithm and Kruskal's Algorithm

For Kruskal's, running Time equals to O(m log n). Testing if an edge creates a cycle can be slow unless a complicated data structure called a "union-find" structure is used. It usually only has to check a small fraction of the edges, but in some cases (like if there was a vertex connected to the graph by only one edge and it was the longest edge) it would have to check all the edges. If the number of edges is kept to a minimum, Kruskal's algorithm works more efficiently.

For Prim's algorithm, if a heap is not used, the run time will be $O(n^2)$ instead of O(m + n log n). However, using a heap complicates the code because of the data structure. A Fibonacci heap is the best kind of heap to use, but again, it complicates the code.

Unlike Kruskal's, Prim's doesn't need to see all of the graph at once. It can deal with it one piece at a time. It also doesn't need to worry if adding an edge will create a cycle since this algorithm deals primarily with the nodes, and not the edges.

For Prim's algorithm the number of nodes needs to be kept to a minimum in addition to the number of edges. For small graphs, the edges matter more, while for large graphs the number of nodes matters more.

Apparently from the given experimental results, Prim's is similar to Kruskal when the graph is sparse (with fewer edges), i.e. example "init", but superior when the graph is dense (with more edges), i.e. example "tb1" or "tb2".

V. Conclusion

Given a set of terminals and a set of obstacles on a plane, two designs of algorithms are proposed to connect these terminals and to avoid running through any obstacle to construct a tree with a possible minimal total wirelength. Design I uses the breadth-first search (BFS) to find the shortest path between any two given terminals and then applies Prim's algorithm for the minimum spanning tree (MST). Design II uses the BFS and then applies Kruskal's algorithm for the MST. Design I and design II achieved the total wirelength of 20 for the example "init", 176 for the test file "tb1", and 238 and 248 for the file "tb2" separately. Theoretical optimality are presented.

Acknowledgment

I would like to acknowledge Prof. Lei He for the classes. I would also like to acknowlege Zhuo Jia, Tianheng Tu and Ke Yi for the design discussions with them.

References

[1] http://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/

[2] http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/

[3] https://www.cse.ust.hk/~dekai/271/notes/L07/L07.pdf

[4] http://en.wikipedia.org/wiki/Breadth-first_search

[5] http://www.ics.uci.edu/~eppstein/161/960206.html

[6] http://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/